Information **2018**, *9*(1), 17; https://doi.org/10.3390/info9010017

Article

CSS Preprocessing: Tools and Automation Techniques ‡

by Ricardo Queirós

CRACS/INESC TEC and School of Media Arts and Design, Polytechnic of Porto, Rua D. Sancho I, n.° 981, 4480-876 Vila do Conde, Portugal

† This paper is an extended version of our paper published in the 6th Symposium on Languages, Applications and Technologies (SLATE 2017), Porto, Portugal, 26–27 June 2017.

Received: 12 November 2017 / Accepted: 10 January 2018 / Published: 12 January 2018

Abstract: Cascading Style Sheets (CSS) is a W3C specification for a style sheet language used for describing the presentation of a document written in a markup language, more precisely, for styling Web documents. However, in the last few years, the landscape for CSS development has changed dramatically with the appearance of several languages and tools aiming to help developers build clean, modular and performance-aware CSS. These new approaches give developers mechanisms to preprocess CSS rules through the use of programming constructs, defined as CSS preprocessors, with the ultimate goal to bring those missing constructs to the CSS realm and to foster stylesheets structured programming. At the same time, a new set of tools appeared, defined as postprocessors, for extension and automation purposes covering a broad set of features ranging from identifying unused and duplicate code to applying vendor prefixes. With all these tools and techniques in hands, developers need to provide a consistent workflow to foster CSS modular coding. This paper aims to present an introductory survey on the CSS processors. The survey gathers information on a specific set of processors, categorizes them and compares their features regarding a set of predefined criteria such as: maturity, coverage and performance. Finally, we propose a basic set of best practices in order to setup a simple and pragmatic styling code workflow.

Keywords: CSS; preprocessors; postprocessors; CSS workflow; web formatting

1. Introduction

Nowadays, when it comes to Web front-end development, we need to understand three languages with different purposes: HyperText Markup Language (HTML) for structuring content, Cascading Style Sheets (CSS) to style it and JavaScript to attach some behavior. Regarding CSS, several surveys show that this W3C specification is used in more than 95% of the websites [1], revealing its enormous importance in the process of constructing a website. A CSS file organizes a set of styling rules in selectors that has the responsibility to select the elements of the target documents that should be styled. Although it is powerful, it lacks many programming constructs such as variables, conditional blocks and functions. This absence leads to CSS developers copying style declarations from one selector to another (e.g., code cloning), fostering code duplication and harming code maintenance. In a previous work [2], the CSS code of 38 high traffic websites were analyzed, and it was found that, on average, more than 60% of the CSS declarations were duplicated across at least two selectors.

In order to address these kind of issues, new tools appeared in recent years, defined as CSS processors and with a common goal: to narrow the gap on the CSS limitations for a modern CSS workflow management. These tools were divided into two big groups: CSS preprocessors and CSS postprocessors. The former are primarily intended to make authoring CSS more dynamic, organized and productive. The latter also affect the web development workflow; however, they operate on the other side of CSS development ("post" development) for automation and refinement purposes.

CSS preprocessors were the first to appear. The code written in a CSS preprocessor can include programming constructs and, thereafter, be compiled to pure standardized CSS. Currently, there are several CSS preprocessors (e.g., SASS [3], LESS [4], Stylus [5]), and their use is becoming a fast growing trend in the industry. In 2012, an online survey [6] with more than 13,000 responses from Web developers, conducted by a famous website focusing on CSS development, showed that 54% of the respondents use a CSS preprocessor in their development tasks. Despite these promising numbers, there are many developers that understand the power of CSS preprocessors but do not know which preprocessor they should use.

CSS postprocessors appeared immediately afterwards. In the CSS workflow, post-processing happens after you already produced the plain CSS, and want to extend it further through automation. This can include extending class selectors, using the new CSS specifications without cross-browser issues or auto-appending prefixes for certain CSS properties. In this context, PostCSS [7] is assuming a crucial role as a tool for transforming styles with JavaScript plugins. These plugins can check the CSS for errors, support variables and mixins, transpile future CSS syntax, inline images, and many other features.

This paper is an extension of a previous conference paper [8], where the author presented a survey on CSS preprocessors. In this paper, the goal is to present tools and automation techniques for the CSS management. With this goal in mind, we start by studying the CSS preprocessors and postprocessor tools by making a simple survey on CSS preprocessors. For this study, we start by selecting active preprocessors with a reasonable amount of popularity and documentation. Then, we describe the selected preprocessors and we compare them regarding a set of predefined criteria such as: maturity, coverage and performance. For the

first criteria, we based our study on the preprocessors' activity in the Git repository hosting service, called GitHub, where the number of commits, releases, contributors and forks are enumerated and analyzed. For the coverage criteria, we rely on a set of features (e.g., variables, mixins, conditional, loops) that developers expect to have in such type of tools and check if these features are supported in the selected preprocessors. Finally, in the performance criteria, we conducted a simple experiment to test the performance of the selected preprocessors measuring their compilation time for a specific set of styling rules.

These results will help in the design of a simple CSS workflow for the modern Web development that gathers pre and post processors in a task runner pipeline. The main goal of using this kind of infrastructure is to free developers of their daily life bureaucratic tasks and concentrate solely in the implementation logic.

The remainder of this paper is organized as follows: **Section 2** introduces the CSS processing subject and distinguishes the two approaches: pre and post processing. In the following section, we initiate the preprocessors survey based on three criteria: maturity, coverage and performance. Next, we produce a set of good practices based on the survey results. Finally, we conclude with a summary of the main contributions of this work.

2. CSS Processors

CSS was written for web designers with limited programming background. Thus, it lacks several programming constructs, such as variables, conditional and repetitive blocks, and functions. This absence affects code reuse negatively, and, consequently, the maintenance of the styling code. These issues were the main reason for the appearance of preprocessor tools such as SASS and LESS. These tools allow developers to use variables, loops, functions, and mixins within CSS. This almost makes basic CSS development similar to programming with extended functionality. However, this was not enough and other tools were launched to deal with other type of issues, mainly related to CSS optimization, automation and to solve cross-browser issues. Thus, after the production of the plain CSS, developers can extend it further by auto-appending prefixes for certain CSS properties, cleaning and organizing CSS rules, appending polyfills for certain properties, generating image dimensions for background images, and many other features.

In general, preprocessing has its own stylesheet languages (e.g., SASS and LESS), that convert into pure CSS. Post-processing takes that basic plain CSS and applies refinement. All this process (**Figure 1**) is time-consuming and error prone.

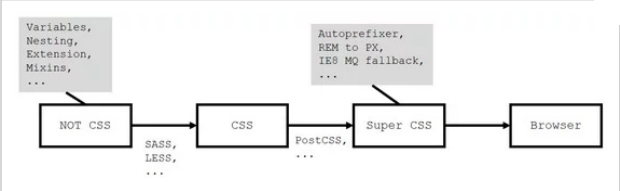


Figure 1. The CSS workflow with the two CSS processor types.

This section is divided into two parts: the first one is dedicated to preprocessors and the second one is dedicated to post-processor tools. We detailed their main features through using the examples of the most popular languages and libraries supported.

2.1. Preprocessors

As previously stated, building a function, or inheritance is hard to achieve using solely CSS. When a web page becomes more complex, we often see CSS files with a lot of rules and a reasonable level of redundancy. One way to save time, and to keep all these rules in a more flexible way, is through the use of CSS preprocessors. These tools make it easy to maintain large, complex style sheets, thanks to the use of features hitherto unavailable in the context of creating style sheets, such as variables, functions, and mixins. Using these constructs, code becomes more organized, allowing developers to work faster and make fewer mistakes. In this section, we analyzed several features typically included in CSS preprocessors. We demonstrate each feature through code examples with both preprocessor and CSS generation code. For the preprocessor language, we use SASS syntax.

2.1.1. Variables

One of the main features of CSS preprocessors is the support for variables. This is a huge feature, since CSS developers lack the chance, for instance, to define a base color as a variable and reuse it all over on the CSS file. Instead, they need hard-coded the hex or named color (or other property such as width, font-size, and others) as a property value for each rule. Using CSS preprocessors, in the case you need to change the color, you only have to do it in one place, in the variable initialization. Using the traditional approach, you must change manually all occurrences of the color. In big and complex web apps, this could be cumbersome and error-prone.

Variables work in a similar fashion to those in any programming language, including the concepts for data types and scope. In addition, like every programming language, preprocessors have different syntax to represent variables' declaration and initialization. Despite their differences, their syntax is like classic CSS, so the learning curve for CSS developers is small. For instance, variables in SASS start with $ sign, in LESS with the @ sign and they have no prefix in Stylus. Both in SASS and LESS, values are assigned with colon (:), and with equals sign (=) in Stylus. In the first example, we declare and initialize a variable and use it as a property value for two rules. The generated CSS will override all the variable occurrences with its value.

Listing 1: SASS code

```
$primaryColor: #eeccff;
body {
  background: $primaryColor;
}
p {
  color: $primaryColor;
}
```

Listing 2: Generated CSS code

```
body {
  background: #eeccff;
}
p {
  color: #eeccff;
}
```

Beyond using variables in property values, it is also possible to use variables in selectors or property names through interpolation. The next example shows how to use variables' interpolation to define a name for a selector and a property:

Listing 3: SASS code

```
$name: foo;
$attr: border;
p.#{$name} {
  #{$attr}-color: blue;
}
```

Listing 4: Generated CSS code

```
p.foo {
  border-color: blue;
}
```

2.1.2. Nesting

As opposed to HTML, CSS nesting structure is hard to write and to understand. Using preprocessors, we can combine various CSS rules by creating composite selectors. Basic nesting refers to the ability to have a declaration within another declaration. With the preprocessors' nesting syntax, developers can organize stylesheets in a way that resembles HTML more closely, thus reducing the chance of CSS conflicts.

A classic example is when we have to style an HTML list composed by a set of links. In this case, we have a structure with several depth levels (<ul>, <li>, <a>) and that requires some care when styling it through CSS rules. Using preprocessors, we have a more intuitive and compact way of styling these types of scenarios through nested declarations. The following example illustrates this situation:

Listing 5: SASS code

```
ul {
  list-style: none;
  li {
    padding: 15px;
    display: inline-block;
    a {
      text-decoration: none;
      font-size: 16px;
      color: #444;
    }
  }
}
```

Listing 6: Generated CSS code

```
ul {
  list-style: none;
}
ul li {
  padding: 15px;
  display: inline-block;
}
ul li a {
  text-decoration: none;
  font-size: 16px;
  color: #444;
}
```

In the case we want to reference the parent element in nested declarations, the & symbol should be used. The next example shows how to add pseudo-selectors using this technique:

Listing 7: SASS code

```
a.myAnchor {
 color: blue;
 &:hover {
  text-decoration: underline;
 }
 &:visited {
  color: purple;
 }
}
```

Listing 8: Generated CSS code

```
a.myAnchor {
 color: blue;
}
a.myAnchor:hover {
 text-decoration: underline;
}
a.myAnchor:visited {
 color: purple;
}
```

2.1.3. Extend

Inheritance is one of the key concepts in the object-oriented programming paradigm. In Object Oriented Programming (OOP) the concept of inheritance provides the idea of reusability. This means that we can add new features to an existing class without modifying it. This is possible by deriving a new class from the existing one. In the CSS realm, inheritance should be used when we need similar styled elements, but requiring slight changes between them. In SASS, we use the keyword @extend following by the base rule name we want to inherit. A classic example is the definition of two buttons: one for confirmation and one for cancellation.

Listing 9: SASS code

```
.dialog-btn {
  box-sizing: border-box;
  color: #ffffff;
  box-shadow: 0 1px 1px 0
   rgba(0, 0, 0, 0.12);
}
.confirm {
  @extend .dialog-btn;
  background-color: #87bae1;
  float: left;
}
.cancel {
  @extend .dialog-btn;
  background-color: #e4749e;
  float: right;
}
```

Listing 10: Generated CSS code

```
.dialog-btn, .confirm, .cancel {
  box-sizing: border-box;
  color: #ffffff;
  box-shadow: 0 1px 1px 0
   rgba(0, 0, 0, 0.12);
}
.confirm {
  background-color: #87bae1;
  float: left;
}
.cancel {
  background-color: #e4749e;
  float: right;
}
```

In the previous example, we note the CSS generated by SASS combined the selectors instead of repeating the same statements systematically, saving us precious memory.

In certain situations, you need to define styles that you just want to be extended and never used directly. A good example is when writing a library (such as SASS), where you just want to provide styles for users to extend in case they need to or ignore otherwise. For this type of scenario, SASS supports placeholder selectors (or placeholders). This type of selectors resemble the class and id selectors (. or #), However, in this case, the % symbol is used. The selectors can be used anywhere and will prevent rule sets from being rendered to CSS, except by extension.

In this example, the %input-style rule will not be generated for CSS. However, if we extend this rule, we will see that it is already possible to obtain the rule in the generated CSS.

Listing 11: SASS code

```
%input-style {
  font-size: 14px;
}
input {
  @extend %input-style;
  color: black;
}
```

Listing 12: Generated CSS code

```
input {
  font-size: 14px;
}
input {
  color: black;
}
```

2.1.4. Mixins

Mixins can be seen as a simplified version of constructor functions in JavaScript language. In fact, it may include styles in the same way that @extend does but as the ability to provide arguments, thus making it a valuable tool for dynamic code reuse and redundancy reduction. The @mixin directive is used to create mixins and the @include directive is used to invoke them. In the next example, a mixin is defined to render squares with colors and sizes passed as arguments to the mixin:

Listing 13: SASS code

```
@mixin square($size, $color) {
  width: $size;
  height: $size;
  background-color: $color;
}
.small-blue-square {
  @include square(20px,
   rgb(0,0,255));
}
.big-red-square {
  @include square(300px,
   rgb(255,0,0));
}
```

Listing 14: Generated CSS code

```
.small-blue-square {
  width: 20px;
  height: 20px;
  background-color: blue;
}
.big-red-square {
  width: 300px;
  height: 300px;
  background-color: red;
}
```

Another typical example is when a property requires prefixes to work in all browsers, such as the transform property:

Listing 15: SASS code

```
@mixin transform-tilt($tilt) {
  -webkit-transform: $tilt;
  ms-transform: $tilt;
  transform: $tilt;
}
table:hover {
  $tilt: rotate(15deg);
  @include transform-tilt($tilt);
}
```

Listing 16: Generated CSS code

```
table:hover {
  -webkit-transform:
    rotate(15deg);
  -ms-transform: rotate(15deg);
  transform: rotate(15deg);
}
```

2.1.5. Functions

Typically, preprocessors support a long list of predefined functions. They serve all sorts of purposes, including string manipulation, color-related operations, and some useful mathematical methods such as random and round. The next example uses the *darken*function that dims a particular color by a certain percentage:

Listing 17: SASS code

```
$my-blue: #2196F3;

a {
  padding: 10px 15px;
  background-color: $my-blue;
}
a:hover {
  background-color:
    darken($my-blue,10%);
}
```

Listing 18: Generated CSS code

```
a {
  padding: 10px 15px;
  background-color: #2196F3;
}
a:hover {
  background-color: #0c7cd5;
}
```

You can also define your own functions and invoke them anywhere through role directives. In SASS, the role directives are similar to the mixins, but instead of returning a set of properties, they return values through the @return directive. The definition of a function is done with the @function directive. The next example shows how to define a function for calculating the width of a column based on the width of its parent, the number of columns, and the margins size:

Listing 19: SASS code

```
SASS code
$ctn-width: 100%;
$col-count: 4;
$mgin: 1%;
@function getColW($w, $cols, $mgin){
  @return ($w / $cols) - ($mgin * 2);
}
.container {
  width: $ctn-width;
}
.column {
  width: getColW($ctn-width,$col-count,$mgin);
  height: 200px;
}
```

Listing 20: CSS code

```
.container {
  width: 100%;
}
.column {
  width: 23%;
  height: 200px;
}
```

2.1.6. Conditional and Cyclic Structures

Some preprocessors support decision and cyclic structures. You can actually use directives like @if, @for, @each, and @while . The next two examples show how to use these directives in two different contexts: decision-making at a given value and iteration under variables to create similar statements. In the first example, the value of a variable is inspected and a set of actions is executed by its value. Note that if the condition in the @if directive evaluates to false, the set of expressions that follow the @elsedirective is executed.

Listing 21: SASS code

```
$boolean: true !default;
@mixin foo() {
  @debug "$boolean is #{$boolean}";
  @if $boolean {
    display: block;
  }
  @else {
    display: none;
  }
}
.some-selector {
  @include foo();
}
```

Listing 22: CSS code

```
.some-selector {
  display: block;
}
```

The following example uses the @for directive to define multiple style declarations named by interpolation:

Listing 23: SASS code

```
$squareC: 3;
@for $i from 1 through $squareC {
  #square-#{$i} {
    background-color: red;
    width: 50px * $i;
    height: 120px / $i;
  }
}
```

Listing 24: CSS code

```
#square-1 {
  background-color: red;
  width: 50px;
  height: 120px;
}
#square-2 {
  background-color: red;
  width: 100px;
  height: 60px;
}
#square-3 {
  background-color: red;
  width: 150px;
  height: 40px;
}
```

2.2. PostProcessors

With the evolution of browsers and the advent of preprocessing tools, several needs appeared that were more related with optimization and browser compatibility. In order to cover all the CSS workflow needs, several tools and libraries were born, acting in the plain CSS files and coined as postprocessing tools, such as, CSSNext, Pleeease, Bless and StyleCow. However, the existence of several tools, nowadays, one tool receives all the attention—**PostCSS**—a tool based on Javascript plugins for transforming CSS. PostCSS, as a CSS processor tool (regardless of the "side") is often compared with SASS. The main advantages are:

- Speed—PostCSS is faster than SASS (even with the appearance of LibSASS),
- Modularity—reduces bloat since you only include the functionality that you need,
- Implementation—developers can immediately implement a new functionality, rather than wait for SASS update.

However, there are still drawbacks regarding (1) Complexity—more planning is required (e.g., plugins must be called in a specific order); (2) Syntax—A different syntax (compared to SASS) and (3) Processing—PostCSS processing requires valid CSS. In the next subsections, we detail the most interesting features supported in the postprocessing workflow. For each feature, we exemplify its use through a PostCSS plugin.

2.2.1. Addition of Vendor Prefixes

Adding vendor CSS prefixes by hand can be tedious and error-prone. In order to facilitate the addition of browser prefixes, the PostCSS plugin—called Autoprefixer—was created. The plugin parses the CSS and automatically adds vendor prefixes to CSS rules using the "Can I Use" database (**https://caniuse.com/**) to determine which prefixes are needed.

This way, developers can concentrate in writing normal CSS according to the latest W3C specs and rely on the PostCSS engine to inject prefixes when necessary. The next example shows the injection of browser prefixes for the display property that specifies the type of box used for an HTML element.

Listing 25: CSS code

```
a {
    display: flex;
}
```

Listing 26: (Super) CSS code

```
a {
    display: -webkit-box;
    display: -webkit-flex;
    display: -ms-flexbox;
    display: flex
}
```

Autoprefixer also removes old, unnecessary prefixes from your CSS (like border-radius prefixes, produced by many CSS libraries). The plugin uses Browserslist (a library to share target browsers between different front-end tools), allowing developers to specify the browsers they want to target in the project by queries like "last 2 versions" or ">5%".

2.2.2. Future-Proof CSS

Beyond the Autoprefixer plugin, PostCSS offers other interesting plugins. One notable example is CSSNext. The plugin transforms CSS specs in standard CSS syntax not implemented yet in browsers (custom properties, custom media, color functions,

etc). The following example depicts those features.

Listing 27: CSS code

```
:root {
  - -red: #d33;
}
a {
  &:hover {
    color: color(var(--red) a(54%));
  }
}
```

Listing 28: (Super) CSS code

```
a:hover {
  color: #dd3333;
  color: rgba(221, 51, 51, 0.54);
}
```

In conclusion, CSSNext is a plugin that allows you to use the latest CSS syntax today, such as the CSS' new custom properties, for example, without worrying about the browser support.

2.2.3. Code Lint

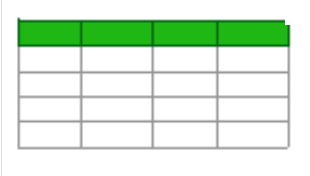
Another interesting plugin is Stylelint, a linter tool to enforce consistent conventions and avoid errors in CSS stylesheets. It supports the latest CSS syntax, as well as CSS-like syntaxes, such as SCSS. The plugin is based on rules that determine what the linter looks for and complains about. The following list enumerates some of the most known rules:

- invalid hex colors,
- duplicate font family names,
- unknown units,
- duplicate properties within declaration blocks,
- unknown pseudo-class and pseudo-element selectors.

3. CSS Preprocessors Survey

This section presents a survey on CSS preprocessors. Currently, there are several CSS preprocessors offering similar features with a different syntax. In order to select one, users typically start by reading Internet sources that describe and compare a subset of them based on an ad hoc list of advantages/disadvantages. This approach could be dangerous since you could invest time in one preprocessor and discover later that is not well documented or do not have an active developers community, or do not support a specific language binding or feature, or even, has poor performance while running in a production environment. This work aims to offer a consistent study to base a sustained choice of a CSS preprocessor. In order to accomplish this challenge, we start by selecting a specific set of preprocessors based on a very popular 2016 front-end tooling survey [9] where 4715 developers (mostly people with five to ten years of front-end technologies experience) answered questions about Web tools and standards. Analyzing deeply the survey, we can find the responses to the question "What is your CSS Processing tool of choice?". The results were conclusive as shown in Table 1.

Table 1. CSS Processing tool.



Analyzing the results, SASS is still the CSS processing tool of choice for the majority of respondents with 63.39%. When compared to last years' results, LESS usage has dropped from 15.19% to 10.14% (down 5.05%).

In addition, we can state that the percentage of respondents using CSS processing tools grows to 86.36% (more 1.4% from 2015). This reinforces the importance of CSS processing tools in the Web development workflows.

We have included tools that are not pure preprocessors, such as PostCSS, which is coined as a CSS postprocessor tool. These types of tools are becoming essential for front-end developers since they can apply specific actions after the CSS has been generated. Among the supported actions, the most popular are applying vendor prefixes automatically, creating pixel and IE8 media

query fallbacks. The most notable example of these kinds of tools is PostCSS. In this survey, one can notice that PostCSS is used by 8.31% of respondents. Actually, its usage is likely to be slightly higher, as this survey does not count for those respondents who use it in combination with another processing tool.

Based on this survey, we select SASS, LESS and Stylus for the CSS Processors survey. In the next subsections, the three preprocessors are described and compared based on three facets: maturity, coverage and performance.

3.1. Maturity

It is difficult to determine which preprocessor is most widely used or has more impact. In this study, we made an effort to measure the popularity/impact of the CSS preprocessors in the Web. Popularity is usually influenced by an active community. When choosing a specific language, library or framework for your project, consider first looking at the community behind and check if it is actively developed for bug fixes and new features. A large and active community will also make it easier to get support and will provide more learning advice and resources. Beyond that, verify how detailed, well-written and organized the documentation is.

Various methods of measuring tools popularity have been proposed, each subject to a different bias over what is measured. In this context, we will focus on two: counting the number of times the language name is mentioned in web searches (using Google Trends) and comparing the activity in GitHub where all the selected preprocessors are stored.

Firstly, we search in Google Trends for the three CSS preprocessors. The results are shown in Figure 2.

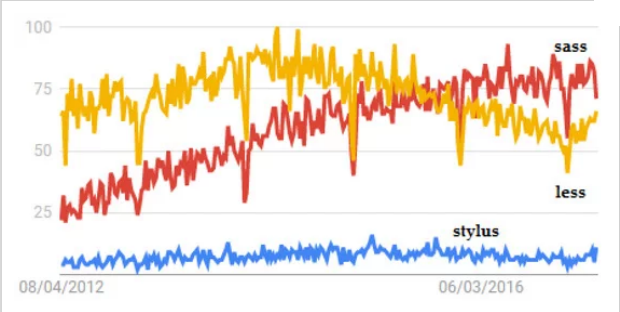
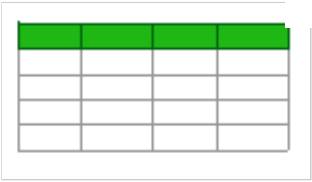


Figure 2. Interest over time on SASS, LESS and Stylus preprocessors—Google Trends.

It appears there is much more activity with SASS and LESS, rather than Stylus. From 2015, SASS is the CSS preprocessor most typed in Google searches.

The second method chosen to measure the impact/popularity of the selected preprocessors was the analysis of their Github activity. In fact, the three preprocessors chosen are pretty active on Github, as you can see in Table 2.

Table 2. CSS Processors Maturity.



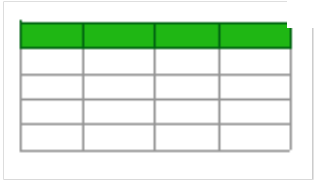
Although relatively recent, these initiatives have been growing with the evolution of the HTML and CSS Web standards. SASS is the oldest and the one with greater update frequency of commits and releases. However, LESS, despite being younger, is the most popular, with a larger number of stars and forks. The number of forks is relevant. A fork is a copy of a repository. Forking a repository allows you to freely experiment a repo (with changes) without affecting the original project. Thus, this means that most people are using LESS base code to start their own projects. Regarding Stylus, it presents the lower values of the three in almost all the indicators.

3.2. Coverage

In the coverage criteria, we will make a comparison of the three CSS preprocessors regarding the support for the most popular features, such as variables, mixins, conditionals and loops.

In Table 3 we present the comparison on CSS preprocessors variable features.

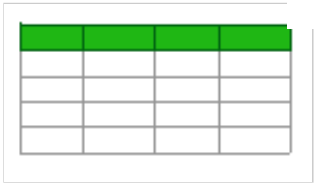
Table 3. Variables features comparison.



All the preprocessors have the basic ability to declare variables and use them later. However, LESS does not support the feature of default variables, that is, variables that are overwritten by regular ones, no matter when they are declared. Nevertheless, variable hoisting (lazy) is only supported by LESS, where variables can be declared after being used. The Stylus preprocessor is the only one that allows you to use the value of a previously declared property elsewhere (variables lookup). Finally, all the preprocessors support interpolation. This means that we can use variables as parts of selectors, properties, values, as parameters of the *calc* function—a CSS function that performs calculations to determine CSS property values—and even place a set of selectors into a variable and reuse it.

Mixins is another powerful feature of CSS preprocessors. The CSS preprocessors mixins support is shown in Table 4.

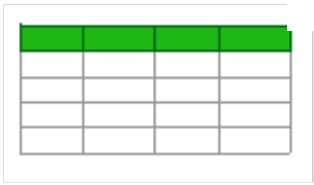
Table 4. Mixins' features comparison.



All the CSS preprocessors support the mixins' main features, namely, the inclusion of mixins (basic), mixins that can receive parameters passed to them (params), mixins that have named placeholders for each parameter passed to them (params-named) and, finally, mixins with an unknown number of arguments passed to them (arguments).

Another important feature of CSS preprocessors is the support for conditionals. Conditionals are useful when we want a part of our code to be executed only if certain conditions are evaluated as true (or false). Although LESS uses a slightly unconventional syntax (*), all the preprocessors support if statements within a declaration. Ternary operators allow for a single-line if/else test in the format of $x > 0 ? \text{true} : \text{false}$. LESS do not support this feature and SASS uses an unusual syntax. Regarding the capacity to interpolate "if statements" inside property names, only SASS and Stylus support it. Table 5 summarizes all the conditional supported features.

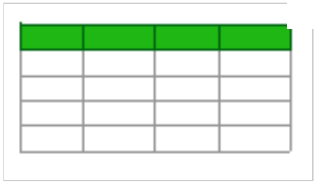
Table 5. Conditional features comparison.



The character * indicates that the preprocessor supports the feature but relies on unusual or unconventional syntax to achieve it.

Regarding loops (Table 6), we only compare loops that can increment values (basic) and loops that iterate though items in a list (intermediate).

Table 6. Loop features' comparison.



All preprocessors support both. However, LESS needs to call the function as shown in the next piece of code:

Listing 29: LESS loop example

```
.generate-column(@i: 1) when (@i =< 4) {
  .col-@{i} {
    width: @i * (100% / 4);
  }
  .generate-column(@i + 1);
}
```

Based on this study, one can conclude that SASS and Stylus are the preprocessors with greater support for the most popular features. LESS has support for conditional and loops, but relies mostly on unusual syntax to achieve it.

3.3. Performance

In this subsection, the three preprocessors are compared in terms of performance. This performance benchmark will be achieved by measuring the compilation time of the preprocessors for different sizes of files.

For the experiment, we started by installing Node.js v6.10.2 (includes npm 3.10.10) in a machine running Windows 10 (64 bits), Intel Core i7-6700K—4.00 GHz, 16 GB RAM and 256 GB Solid State Drive (SSD).

For the compilation process, we used the preprocessor compilers for Node.js, more precisely, nose-sass, node-less and node-stylus.

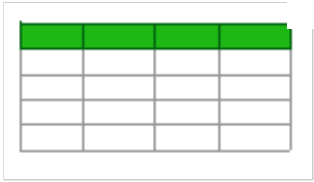
The next step was the creation of a JavaScript file with the test code. The aim of this test is to compare CSS preprocessors for parsings, nested rules, mixins, variables and math. An excerpt of the test is presented in the following code:

Listing 30: Benchmark script excerpt

```
var path = require('path');
var fs   = require('fs');
var n = 999;

// SASS
var libsass = require('node-sass');

var scss = '$size: 100px;';
scss += '@mixin icon { width: 16px; height: 16px; }';
for ( i = 0; i < n; i++ ) {
    scss += 'body { h1 { a { color: black; } } }';
    scss += 'h2 { width: $size; }';
    scss += 'h1 { width: 2 * $size; }';
    scss += '.search { fill: black; @include icon; }';
}

var result =  libsass.renderSync({ data: scss });
console.log("SASS: ", result.stats.duration + "\t ms");

// Repeat for the other two preprocessors
...
```

The test starts by the generation of the preprocessor code of three sizes (10/100/1000 KB). Then, it uses the internal function of each library for the compilation of the preprocessor code to the CSS format. The compilation process for each library is measured and the processing time are showed in the console. The results are presented in Table 7.

Table 7. Performance benchmark.



Based on these results, the main conclusion is that SASS is the fastest of the three. Please note that the official implementation of SASS is open-source and coded in Ruby, but we did not find any node compiler based on this implementation. For these tests, we used Node-SASS, a library that provides binding for Node.js to LibSASS, an open source C++ implementation of SASS. Thus, you should take into account that the results may appear a bit inflated in favor of sass.

4. CSS Workflow Best Practices

After this in-depth study of CSS processors, we came to the conclusion that CSS processing can be done on two sides: preprocessing allows the developer to write and maintain cleaner and more modular CSS code using constructs closer to the programming languages; postprocessing allows the developer to optimize and clean up their stylesheets, to use the latest specifications without worrying about multi-browser support and other type of refinements. From both sides, two tools stands out: the SASS in the preprocessing side and the PostCSS in the post-processing side. Despite this division, the trend is clearly for the merging of these two types of processing, since most of these features can be implemented in both sides. At the same time, CSS specification is evolving and supporting some concepts of these processing tools such as variables through the concept of custom properties. This evolution makes it even more difficult to estimate what the future of these technologies will be.

In order to create a CSS workflow, it is necessary to understand what the development lifecycle of a modern Web application is.

4.1. Web Development Lifecycle

Nowadays, Web applications are becoming increasingly complex. Developers no longer write ad hoc JavaScript and jQuery, instead they use a lot of frameworks, libraries and languages that should be managed efficiently. In this context, developers, when building a web app and for improving productivity and satisfaction, are starting to use front-end development workflows (Figure 3). These workflows typically comprise three types of tools:

1. scaffolding tools,
2. build tools,
3. package managers.

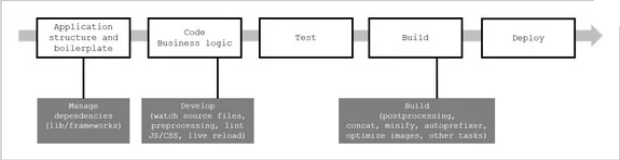


Figure 3. Web application's development lifecycle.

A **scaffolding tool** involves the creation of the application structure and the addition of boilerplate code to allow developers to get started and follow the best practices (popular examples: Yeoman and Brunch). The **Build System** is used to build, preview and test the project (popular examples: Grunt and Gulp). The **Package Manager** is used for dependency management. Using this kind of tools, developers no longer have to manually download and manage their scripts (popular examples: npm and Bower).

4.2. Design of a Modern CSS Workflow

Based on the previous surveys and workflows, we propose a simple architecture for a CSS workflow (Figure 4) that includes both SASS and PostCSS processors as well other common tools that belong to the daily life of all front-end web developers. Rather than exhibit all the abstract details of the proposed architecture, we present a simple practical example (Figure 4). In this example, we use SASS scripts to write the styles for the Web application. Then, we compile plain CSS files. These files will be merged by the package gulp-concat-css, bubbling up *@import* statements. Finally, we use the PostCSS plugin CSSNext (via Gulp) that allows for using the new CSS syntax and transforming it into a more browser-compatible CSS so developers don't need to wait for browser support. This finishes the workflow tasks and allows the browser to interpret all the HTML/CSS/JavaScript files and to handle user interaction.

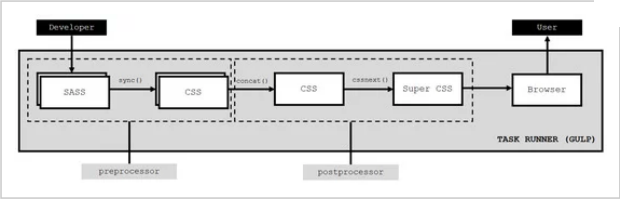


Figure 4. The CSS workflow with the two CSS processor types.

At the heart of this workflow is the build system as a task runner with the ultimate goal of automating all the front-end tasks so that developers can concentrate on the Web app logic. In this practical example, we used Gulp as the task runner. An example of using PostCSS along with SASS and Gulp is as simple as the following code snippet:

Listing 31: Using PostCSS along with SASS and Gulp

```
var gulp = require('gulp'),
    concatcss = require('gulp-concat-css'),
    sass = require('gulp-sass'),
    postcss = require('gulp-postcss'),
    cssnext = require('postcss-cssnext');
gulp.task('stylesheets', function () {
  return (
    gulp.src('./src/css/**/*.scss')
    .pipe(sass.sync().on('error', sass.logError))
    .pipe(concatcss('all.css'))
    .pipe(postcss([cssNext()]))
    .pipe(gulp.dest('./dist/css'))
  )
});
```

The snippet code above starts by storing references to all of the needed modules (Gulp, Concat CSS, SASS, PostCSS, and CSSNext) in a series of variables. Then, it registers a new Gulp task called stylesheets. This task watches for files that are in ./src/css/ with the extension .scss (regardless of how deep in the subdirectory structure they are), SASS compiles them, and concatenates all of them to a single all.css file. Once the all.css file is generated, it is passed to PostCSS to transpile all of the PostCSS (and plugin) related code to the actual CSS, and then the resulting file is placed in ./dist/css.

In conjunction with Gulp, we can use the Browserify tool that will recursively analyze all the require() calls in your app in order to build a bundle you can serve up to the browser in a single <script> tag. Other alternatives are using Grunt or Webpack (a module bundler) in conjunction with Node Package Manager (NPM) scripts.

5. Conclusions

CSS preprocessors are very powerful and they can help streamline your Web development process, especially if you are coming from a programming background. In this paper, we started with a survey to evaluate a set of CSS preprocessors regarding a set of predefined criteria such as: maturity, coverage and performance. While it appears that SASS is more widely used and has better performance, Stylus covers very well the main and typical features of a CSS preprocessor. In fact, there is not really a preprocessor that is better than the other. It usually comes down to the developer and what they are comfortable with using. The most important thing is to use one in order to help you make your CSS more maintainable and extendable.

When it comes to cross-browser fixes like vendor prefixes and other refinements, preprocessors should be avoided. In this case, let postprocessors deal with those issues since they are faster. In this realm, PostCSS is the most popular with more than 200 Javascript plugins that allow developers parse plain CSS files for optimization purposes.

Based on this study, a set of best practices for the use of CSS processors was presented. In this context, we propose an architecture to model a modern CSS workflow based on the concept of task runners, where features of both sides are all chained in a stream pipe. This approach reveals itself to be more flexible, modular and scalable and can be extended for further use. At the end of 2017, concerning task runners and bundles, the Webpack+NPM scripts are competing with Gulp+Browserfy. Despite this competition, the focus is not the task runner and bundle tools, but choosing the right tools for the desired goals and integrating them into a consistent workflow for easy and flexible management.

Acknowledgments

This work was supported by FourEyes—a Research Line within project "TEC4Growth—Pervasive Intelligence, Enhancers and Proofs of Concept with Industrial Impact/NORTE-01- 0145-FEDER-000020" financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF).

Conflicts of Interest

The author declare no conflict of interest.

References

Mazinanian, D.; Tsantalis, N. An Empirical Study on the Use of CSS Preprocessors. In Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Suita, Japan, 14–18 March 2016; Volume 1, pp. 168–178. [Google Scholar]

Mazinaniana, D.; Tsantalis, N.; Mesbah, A. Discovering Refactoring Opportunities in Cascading Style Sheets. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Hong Kong, China, 16–22 November 2014; pp. 496–506. [Google Scholar]

Catlin, H. SASS: Syntactically Awesome Style Sheets. Available online: **http://sass-lang** (accessed on 1 April 2017).

Sellier, A. LESS—The Dynamic Stylesheet Language. Available online: **http://lesscss.org/** (accessed on 1 April 2017).

Holowaychuk, T. STYLUS CSS. Available online: **http://stylus-lang.com/** (accessed on 1 April 2017).

Coyier, C. Popularity of CSS Preprocessors. Available online: **http://css-tricks.com/poll-results-popularity-of-css-preprocessors** (accessed on 11 June 2017).

Sitnik, A. PostCSS. Available online: **http://postcss.org/** (accessed on 1 April 2017).

Queirós, R. A Survey on CSS Preprocessors. In Proceedings of the 6th Symposium on Languages, Applications and Technologies (SLATE 2017), Porto, Portugal, 26–27 June 2017; Queirós, R., Pinto, M., Simões, A., Leal, J.P., Varanda, M.J., Eds.; Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik: Dagstuhl, Germany, 2017; Volume 56. [**Google Scholar**]

Nolan, A. The State of Front-End Tooling 2016—Results. Available online: **https://ashleynolan.co.uk/blog/frontend-tooling-survey-2016-results** (accessed on 29 November 2016).